Package: TMBO (via r-universe)

September 10, 2024

Title Code helpers for TMB

Author M.V. Bravington

Description Arrays/vectors from index 1 (or any offset you choose, like package 'offarray'), names and dimnames, useful non-crashy error messages, plus simple for-loop aliases. Nicer than raw TMB!

Imports tools, mvbutils, offarray, TMB

LinkingTo TMB

SystemRequirements C++20

Maintainer Mark V. Bravington <markb2@summerinsouth.net>

License GPL-2

Version 2.4.29

Repository https://markbravington.r-universe.dev

RemoteUrl https://github.com/markbravington/TMBO

RemoteRef HEAD

RemoteSha c367331d1facff981a2bef3a26a51e1a8d64d573

Contents

TMBO-package *R-style indexing and convenience for TMB*

Description

TMBO helps you simplify your TMB-style [see far below] C++ code to work more easily with R. Your variables can now use 1-based indexing just like in R rather than 0-based, minimizing coding confusion and pointless mental overhead. And if you use my R package **offarray** too, your indices can start at any offset you like, in both R and TMB. You can get nice polite OOB (Out-Of-Bounds) messages in R showing exactly where the problem occurred and with what values, instead of crashing your R session. What's more, R-side `names` and `dimnames` can be respected and propagated back to R; there's a much simpler for-loop syntax for TMB; and there's a neat way of defining "nested" functions, i.e. inside your objective. All these features are optional; regular TMB code will be unaffected. These might sound trivial but believe me they are *not* when you are writing hundreds to thousands of lines of stock-assessment or CKMR code!

Here's some TMBO code:

```
...
#include <TMBO.h>
TMB_MAGIC{
PARAMETER_VECTOR1( parz);          // parz(1) is *first* element !!!
ICONSEQ( Samp_years);              // range from R--- eg 1987 to 2003
CHIND( Sexes);                     // character index range, from R
// Data from R: in this case, an 'offarray' which I happen to know has
// dims Samp_years and Sexes
DATA_MATRIXO( C_ys);
/*
 Declare inside TMBO, instead of vector<Type> totC_s(...)
 In this case, a base-R vector starting
 at 1 (not an offarray), with names
*/
VECTOR1( Type, totC_s, Sexes);
FOR( s, Sexes){                    // so simple!
  Type temp= 0;
  FOR( y, Samp_years)              // or eg FOR( y, (2003, 2008))
    temp += C_ys( y, s);           // natural syntax
  totC_s( s) = temp;
};
REPORT( totC_s);                   // in R you will get eg this:
//  Female    Male
// 11772.3  8711.2
return( 99.99);
};                                 // And they all lived happily ever after
```

TMBO is very easy to use. On the R side, just do `library(TMBO)` instead of `library(TMB)`, to get lookalikes of `MakeADFun`, `compile`, and `runExample`; check their docu for extra TMBO-specific features. On the C side, the syntax extensions are straightforward; you can see many of them above.

TMBO variables that are acquired from R— eg `DATA_ARRAYO( a3d_thing)`— or declared inside TMB—eg `MATRIX1( int, zmat, 7, 11)`— are just TMB variables that work exactly as usual inside TMB, with the *sole* exception of subscripting individual elements in *your own* code. Thus, `zmat.size()` will still be the number of elements in `zmat`, but the very first element is now *referred to* as `zmat(1,1)` and the very last is `zmat(7,11)`. Note that TMB's built-in operations on entire vectors, matrices, and arrays are oblivious to the index offsets. When you `REPORT()` such a TMB variable, the R-side output will be made nice, with any `names`, `dimnames`, and index offsets as appropriate.

To turn on OOB checking, you just need to set the argument `ppflags="-DTMBOOBOO"` when you call `compile`. TMBO can only OOB-check variables that you've declared with TMBO's macros, eg `DATA_VECTORO`, `PARAMETER_MATRIX1`, `ARRAYO`— not ones you've created with `vector<Type>` etc. When you've eliminated OOBs, remember to recompile without the checking, coz it will slow things down otherwise.

For more examples, `runExample()` will list them, and also run them individually or collectively. They are all in the folder `system.file( "examples", package="TMBO")`, so that you can examine the C++ and the (minimal) R sources.

Detailed documentation of the C macros is in `?MACROS`. There is also an FAQ, available via `RShowDoc("TMBO-FAQ",package=` please note its advisory warning at the start! See also `package?offarray`, and note the byte-compiler issue.

One based and general offsets

Many people will just be happy to be able to use 1-based and "named" indexing in TMB, to match R; they don't need to read the rest of this subsection, which is about general index offsets. Just stick to `FOR(i,(1,N))` and `MATRIX1` and `DATA_VECTOR1` etc; that's fine.

But not me; *I* want to start *my* array indices *wherever I prefer*, eg so that `catch[2000]` in R or `catch(2000)` in TMB means the catch in year 2000 (without also storing 2000 unused elements dating back to 1AD!). This is extremely useful in "real-world" biological/population-dynamics modelling situations such as mark-recapture or, especially, stock-assessment; and it's also useful for the many mathematical algorithms that are naturally written in terms of vectors/arrays that start at some other index (e.g. Wynn's epsilon algorithm, starting at "i=-1"). When I have to write complicated (100s to 1000s of lines) code for stock-assessment and particularly Close-Kin Mark-recapture, for which 9-dimensional arrays and loops are not uncommon, I really don't want to have to remember how to shift all my loop indices, or think about whether I need to add 1 to ranges and differences, or... etc. Making "out-by-1" (or out by much more!) mistakes in complicated code is *incredibly common*, and sometimes can be very hard to diagnose.

Therefore, my `offarray` R package (from c. 2018) lets you index arrays etc exactly where you want, as well as speeding-up/simplifying multidimensional loops. I find it indispensible for moderately-complicated R code such as simple CKMR (though the full complexity of eg age-structured length-based fish stock assessment is just too much for R). And the **TMBO** package works seamlessly with `offarray`. So you can have R variables like this:

```
Rprompt> C_ys
        Female  Male
[1987,]  24083 92992
[1988,]  10354  6710
[1989,]  77635 76861
```

where C_ys[1988,"Male"]==6710; and in TMBO you could JUST refer to C_ys(1988,Male_)'. And things coming out of REPORT() will have the correct offsets/names/dimnames in R. And so on.

Functoids

TMB does not make it easy (or, at least, does not actually tell you how) to define "nested" functions that can access/change variables in your objective function. People have sometimes used horrendously complicated things, apparently called "functors", to get round this.

In fact, there is quite an easy way, but it's rather obscure: you just need something that appears to be called a "non-anonymous anonymous lambda function", and which has a very slightly weird declaration syntax that I have prettied up for you; just declare your nested function fun via FUNCTOID(fun)(<args>){<code>}. In view of the completely ludicrous official name, I decided to give these things a nickname instead, hence "functoid". I don't care what you call it. See the example "functoid.cpp".

Functoids are "just" a base feature of C++ anyway, and they don't "belong" particularly in TMBO, except that it is a good place to mention their existence (and I have added a homeopathic dose of syntactic sugar, via the FUNCTOID declaration macro). However, a bit of extra care is required because of TMBO's offset-index extensions, as follows. Because a functoid can already access all the variables in your objective function, you don't usually need to explictly pass in any VECTORO-etc arguments. However, if you do want to, then you need to wrap the argument (say x, a VECTORO of type Type) in USE_VECTORO(Type, x) when declaring the functoid, and then pass in the actual version of x (say y) via PASS_VECTORO(y) when calling the functoid— analogously for MATRIXO, VECTOR1, etc.

The main things to be aware of with functoids, are that:

- you must declare any variables that the functoid will use *before* you define the functoid itself;
- any variables created *within* the functoid will *not* be visible outside it;
- *never* declare an argument to the functoid with the same name as a variable in the objective!

IS.IT.RISKY?

Clearly, TMBO is written by me (MVB) not the TMB core folk, and so you might wonder if you are taking a risk by using it; what if MVB loses interest / gets busy / gets squashed by a bus? What will happen to your code?

It's a fair question, but I would say that TMBO is pretty safe and future-proof, for the following reasons.

1. There is no interference or modification of TMB itself; it's all done by C macros (see below). Thus, even if TMB's internal workings get changed in future, things should still work fine. (If TMB's *interface* changes, your vanilla TMB code would also stop working anyway, regardless of TMBO or not.)

2. The **TMBO** package has (or soon will have) minimal R dependencies, apart from package **TMB** itself. Currently it imports mvbutils and offarray, but I will change that by manually including a few basic mvbutils functions, and making offarray into a "Suggests"; offarray is not required if you are happy to stick to 1-based base-R-like variables.

3. TMBO consists of a *small* amount of R wrapping, and a *large* number of C macros that augment your C++ code before the **TMB** package compiles it. As much as possible is actually done by C macros not R, in the interests of "clarity" and "stability"; the C macro language is not going to

change much, nor disappear. ("Clarity" is perhaps a stretch, because the macros are hard to understand; they use heavy-duty macro trickery from the internet. But the macros certainly are **stable**.) The R wrapping, which mostly lives in `TMBO::MakeADFun`, sets up a few more variables in R that are passed into TMB and referenced in the augmented C++ code, makes some tiny further tweaks to that code (in `compile`), and also (in `TMBO:::new_report`) tweaks the results of `$report()` in R.

3a. Actually, there is a bit of C code to access TMB's internal structures (array elements), but it's only activated when OOB-checking is on.

4. You can always run `compile( <myTMBOfile>, ppcheck=TRUE)` to get legal TMB code that will subsequently work with "vanilla" TMB. It won't look pretty, and you will have to manually add some helper variables to `data=...` in R, to take care of offsets etc, but it will run. Even if you are using general offsets (non-1), you could if desperate get the TMB code to work without `offarray`.

5. I do need to modify `offarray` at some point, to work around a bug in R's byte-compiler, but there's a perfectly functional workaround in the package as it stands. (See **Quirks**.)

Tmb code in packages: `TMBO` is a very new package, and I've only tested it for standalone compilation (and only on Windows, mea culpa), not for incorporating your own TMB code into your own package. Presumably your DESCRIPTION file will need to say "Imports: TMB, TMBO" and your NAMESPACE file will need some simple mods. At present, the only "duplicated" functions from TMB itself are `MakeADFun`, `compile`, and `runExample`, but I might need to add other more-obscure ones over time. To avoid dull warnings, you'll need to *not* import those from TMB itself. See "Writing R extensions" (sorry... there's no way around that!).

Putting TMB code into a package is apparently tricky enough that another helper package, `TMBtools`, is needed. I'm not sure how that will all interact. A further complication with `TMBO` code is the preliminary steps of pre-processing and R tidy-up, which will entail modifications to "Makevars" etc; OMG. AFAICS makefiles are the worst thing in the universe and I am greatly not looking forward to the task, in the strongest possible terms. However, it is definitely do-able, and someone who knows what they are doing could probably get it working in 10mins. Assistance would be welcome!

Dependencies: This is a minor note that should go somewhere, though perhaps not right here! It concerns the dependencies of `TMBO`, in particular `TMB` and `offarray`. The `TMBO` DESCRIPTION file lists those packages in "Imports" not "Depends", to avoid them getting unwantedly attached to the search-path whenever `TMBO` is only used as an import rather than itself being attached (eg in your own packages). However, in normal non-packagey use when you just call `library(TMBO)`, you'll want the functionality of `TMB` and `offarray` to be exposed automatically— but you don't want the `TMB` version of eg `MakeADFun` to accidentally override the `TMBO` version, and you don't want to have to remember to do `library(TMB)` *before* `library(TMBO)`. Thus, `TMBO` has an `.onAttach` hook which will attach `TMB` and `offarray` *below* `TMBO` (so that `TMBO` versions take priority). This only executes when you call `library(TMBO)` explicitly. It should work. Cross yer fingers.

HOW.DOES.TMBO.WORK?

The FOR-loop equivalents (FOR and ROF) are very simple macros that save typing, increase clarity, and reduce the very real possibility of errors: `for( i...; ...; j++)`, anyone?

The main trick in `TMBO` is the index offsets. Basically, when you declare eg `x` as `VECTORO`, that leads to the creation of an "alias" for `x`: a C function-macro with the name `x`, which shifts its argument by an offset before invoking the "real" `x(...)` on that shifted value. Because C does not recursively expand macros (unless devious trickery is used), that "real" `x`-access is left alone. And because a function-macro cannot share a name with an object-macro, any references to `x` that are *not* `x(<subscript(s)>)` are unmodified. The macro alias version of `x` is created by `#define`

when you write eg VECTORO(<sometype>, x, <dimrange>). But because you can't define new macros within a macro definition in C, a second pass of CPP is necessary to make everything work.

Those shifted-subscript macros are fairly simple, but there is more complexity in the creation macros such as VECTOR1 and DATA_ARRAYO, which allow efficient declaration of offset variables (either coming in from R, or declared from new in your TMB code). Those declaration macros have to create the alias macros, but also have to set up some helper variables in TMB (mostly passed in automatically from R) to store the index-offsets and relevant information about character indexing.

There is also a modified REPORT() macro that invisibly reports some extra helper variables, and a modified version of $report in the result of MakeADFun. The latter can stitch together the raw TMB::report output to make nice R-side variables with the correct offsets, names, and/or dimnames.

TMB::MakeADFun inspects its data= and parameters= arguments a bit more carefully, to set up those extra helper variables. It also has a ranges= parameter, where you can define named ranges (integer or character) than can be referred to your TMBO code: for example,

```
ranges= list(
    Samp_years= c( 1987, 2003),
    list( SEXES=c( 'Female', 'Male"))
  )
```

(For convenience, it's usually better to use TMBO_ranges (qv) rather than list, but the idea is the same.) Again, any ranges= result in extra variables passed invisibly via an augmented data argument to TMB::MakeADFun, and referenced in the augmented C++ code. The report function from TMB::MakeADFun is also tweaked so that it automatically calls a modified cleverer version, described next.

WHAT.IS.TMB?

The C++-software/R-package TMB, for automatic differentiation and automatic Laplace approximation (and more) of code that you write in a dialect of C++, is an amazingly powerful and practically indispensible tool for modern statistical modelling. Thank you, Kasper K & co!!! However, it must also be acknowledged that TMB can be quite painful to work with. Some of that seems to be unavoidable side-effects of the intrinsic gruesomeness of templated-C++, but the 0-based indexing restriction is quite fixable. As is the (lack of) error messaging on OOBs. And for-loops in C are just horrible: verbose and error-prone. Here is a genuine example, including a genuine comment, from a colleague's CKMR code (only slightly tweaked by me):

```
// this is one of the worst things I've ever written...
for(int rj=0; rj<REGIONS.size(); rj++){
  for(int ibj=0; ibj<JCOHORTS.size(); ibj++){
    int bj= JCOHORTS( ibj);
    for(int iyj=0; iyj<SAMPY.size(); iyj++){
      int yj = SAMPY( ibj);
      for(int ibc=0; ibc<ACOHORTS.size(); ibc++){
        int bc= ACOHORTS( bc);
        for(int iyc=0; iyc<SAMPY.size(); iyj++){
          int yc= SAMPY( yc);
          for(int rc=0; rc<REGIONS.size(); rc++){
            Type temp= 0;
```

```
                for(int iw=0; iw<MAX_WEANAGE - MIN_WEANAGE; iw++){
                  int w= MIN_WEANAGE + iw;
                  temp += Pr_w(w) *
                      Pr_rr_t( rj , rc, tsep( yj, yc, bj+w));
                } // for w
                Pr_r_by_MOP( rj, ibj, iyj, ibc, iyc, rc) = temp;
              }
            }
          }
        }
      }
    }
```

Even if you are very used to writing code like that, its sheer length and ugliness does not exactly facilitate comprehension nor future maintenance. Oh, and there's at least two typo bugs in there which, unfortunately, compile without errors. Did you spot them?

The TMBO equivalent could be:

```
FOR( rj, REGIONS) FOR( bj, JCOHORTS) FOR( yj, SAMPY)
  FOR( bc, ACOHORTS) FOR( yc, SAMPY) FOR( rc, REGIONS){
    Type temp= 0;
    FOR( w, MIN_WEANAGE, MAX_WEANAGE)
      temp += Pr_w(w) *
          Pr_rr_t( rj, rc, tsep( yj, yc, bj+w));
    Pr_r_by_MOP(rj, bj, yj, bc, yc, rc) = temp;
  };
```

Nuff said! And BTW the opportunity for those typos did not even arise.

Gratuitous observation

Pascal and Fortran programmers are well used to the "luxury" of indices starting wherever you bloody well tell them to— a feature of those languages for over 40 years! (And even C itself will let you do so, if you are mildly devious.) Contrast that with the "advanced modern power" of C++23... :/

Limitations

These are pretty minor...:

1. With TMB (and C++ in general), you can declare a variable and initialize it in a single operation, eg

```
vector<Type> blurble << pi, 2*pi, 4*pi;
```

But with TMBO, you have to first declare offsetty variables in a single step, and only then can you put stuff into them:

```
VECTORO( Type, blurble, (1, 3));
blurble << pi, 2*pi, 4*pi;
```

2. Extractors such as <vector>.segment() <matrix>.row(i) and <matrix>.col(i) and <matrix>.block(...) still use zero-based indices. Although I probably could cook up some macros to fix that (eg ROW(<matrix>,i)) which would work whether <matrix> is TMBO or not), I doubt that those functions are used much in the context of 1-based or general-offset-based work. I'm not sure they even exist in TMB for arrays (though they should).

Quirks

I should mention that offarray (as of version 1.0.183) has a bad interaction with (a bug in) R's byte-compiler— so you have to turn off the byte-compiler to use offarray reliably. That can be done by running a tiny script; see offarray documentation for details. I use offarray all the time, and there are no problems after turning off the BC. At some point I will write some C code to get round it.

One TMBO quirk at the moment is that I've had to do some epic fly-hacking in order to get TMBO's include path picked up by TMB::compile. It does work, but crikey... There are a couple of less devious ways to address that. Arguably the cleanest, would be to make compilation 3-phase rather than 2-phase at present; there would be *two* successive CPP passes using **only** TMBO headers, leading to a completely standard TMB file that's been full macro-expanded WRTO TMBO constructs, ie with no remaining mention of TMBO at all. (The file contents would be pretty ugly, though... but maybe that doesn't matter). The second way I can think of, is to set the TMBO header as an object-macro during invocation of the first CPP pass, and fixing "TMBO.h" to make sure that explicit paths are expanded into the hash-includes for the second pass. The third would be to hack the hash-includes before the second pass, during the R code that tidies up the CPP output. All these alternatives are more work than the fly-hacking, however. I don't think this "quirk" amounts to a risk for the future, because I certainly *could* implement the first or third alternatives, and *probably* could do the second.

---

compile *Compile a TMB CPP file into dyn-load-able, using TMBO features*

Description

Like TMB::compile (qv) but allowing TMBO features: offset arrays (things don't have to start at zero) and greatly simplified for-loops.

The mechanism is to first run the C preprocessor on your file eg "mymod.cpp" to handle *only* the special TMBO macros, producing an intermediate file eg "mymod_TMBOpp1.cpp", then tidy that up with a small amount of regexing in R, then run TMB::compile on it, then rename the result back to "mymod.dll" or "mymod.so".

Your original file needs to have "#include <TMBO.h>" instead of "#include <TMB.h>", otherwise this won't work. Also, any *system* headers (eg of RcppEigen) should be written HASH_INCLUDE <syshead>" not "#include", so that they dont get expanded during the first pass, but only during the second.. However, your *own* headers (eg if you have split your code into various source files) should be "#include" as usual.

Compiler name: If compile works for you out-of-the-box, don't read this bit. If not...

As of version 2.0.x of package **TMBO**, the preprocessing pass defaults to trying to run a program called "cpp", which I thought stood generally for "Chucky's Pre-Processor" ;). It works (for

me...) on Windows, because that program lives in the same folder as "g++" which actually does R's compilation; I think they're synonyms FAPP, so I could instead have called "g++" with "-E" option (only preprocess). But "cpp" might not work on all platforms, and "g++" presumably does not (it's GNU, so maybe not for Macs). So for pre-processing on non-Windows, *you* might need to use a different "program".

As of v2.1.x, I have added some logic in `.onLoad` to *hopefully* autodeduce the right name. But if that doesn't work, you can specify the compiler's name manually *before* calling `compile` (or, better, before loading `TMBO`) eg via `Sys.setenv( CPP_IN_R="clangaroo")` or whatever the magic name should be on your system. If you are a Linucian you probably know exactly what to do. Whatever you supply has to accept arguments "-E" (preprocess only) and "-CC" (include comments in macros), otherwise no go.

The main compilation pass eventually calls `tools:::.shlib_internal` (via `TMB::compile`) which knows what the real compiler is called, so that should be fine. You can actually force `.shlib_internal` to reveal the real compiler's name, so that's what's in my autodeduce code—but it has required some more superhacking, because R has (accidentally, I *hope*) made it *far more difficult* than it needs to be. Sigh. Again.

Usage

```
compile( file, ..., dev= exists( '..TMBO', mode='environment'),
  ppcheck= FALSE,
  stop_after_cpp1= FALSE,
  ppflags= NULL,
  flags= NULL
)
```

Arguments

`file`	The main source file, optionally with path. Extension ".cpp" will be automatically added if needed..
`...`	Passed to `TMB::compile` (qv). See also `flags` argument.
`dev`	Leave this alone, it's just for me.
`ppcheck`	if TRUE, do 2 passes of preprocessing and tidy-up, but then stop before normal TMB compilation. The output is returned as a character vector, showing pretty much what `TMB::compile` would see. Maybe helpful if you are getting incomprehensible compiler errors. Note that the *first* round of preprocessing and tidy-up always gets stored in an intermediate file, regardless of `ppcheck`; see **Value**.
`stop_after_cpp1`	surely this is self-explanatory? You'll just get a file called "<file>_TMBOpp1.cpp", before the R-side tidy-up (which is essential before the second pass).
`ppflags`	optional character vector of flags to give to the first preprocessor pass, eg `-DTMBOOBOOB` (when that is working...)
`flags`	optional string, or character vector which will be concatenated into a string, that gets passed to `TMB::compile`. Probably flags of some kind used for something; the documentation has not enlightened me... On Windows, you might want to use eg `flags="&> logfile.log"`, so that any compilation errors are sent to that file.

Value

Various files are produced (see below). From a purely R PoV, though, the return value shows the outcome of the compilation attempts: 0 (success) or 1 (failure). That's the usual convention, strange as it may seem: so `if(compile(...))` actually means "if compilation fails". If `ppcheck=TRUE`, no compilation per se is attempted, just two rounds of preprocessing. The "normal" error case is that `ppcheck` is FALSE, preprocessing (first pass) succeeds, but compilation/second-preprocessing-pass fails. In that case, the logfile contains the compiler's error log (if you remembered to set a logfile via `...`). If `ppcheck=TRUE`, then the result will have attribute `pplog`, a character vector containing (if succhess) what TMB itself would see, after expanding all the TMBO macros. If preprocessing fails (in the first pass regardless of `ppcheck`, or just in the second pass if `ppcheck` is TRUE) then the result will again have attribute `pplog` showing the complaints. That's the plan, anyway: IDK if all 3 types of error return exactly what they're supposed to, especially not on Linux. The files produced go into the same folder as `file` itself. The first is an intermediate C++ file "<myfile>_TMBOpp1.cpp", arising from the first preprocessing pass. If that pass gives an error, that intermediate file may be incomplete. If successful, the intermediate file is then tidied up a bit in R and saved back under the same name before considering `ppcheck`, so you never get *pure* successful CPP output unless you have set `stop_after_cpp1=TRUE`. If full compilation succeeds, you should get a DLL (aka "shared-object file") with the same name as . If compilation fails and you remembered to ask for a logfile, there will be one; good luck deciphering it... Probably there's some kinda ".o" files representing some kinda C-level muckery, too.

Note

As the in-code comments say, there is some spectacular hacking inside `TMBO::compile`, to ensure that TMBO's include-path gets seen by `TMB::compile`. The latter calls `tools:::.shlib_internal` at some point, so the trick is to intercept that and kludge the desired path into the makefile... ugggh. It works fine, but there really should be a better way! TMB folk may be able to suggest one.

These further0 notes are really just for me. One TMBO-only alternative would be 3-phase compilation, with the 2nd phase being a pure CPP run on `pp_full_file` allowing **only** <TMBO2.h> as the include file. That's close to what `ppcheck=TRUE` does now, but seems Ugly.

Another **might** be to define a macro during the first call to CPP that contains the TMBO include path; this would then be incorporated into the HASH_INCLUDES that get added for the 2nd pass; the macro would give the full include path for "TMBO2.h" (and "boring_array_bits2.h" and ...).

Examples

```
## Not run:
## Normally use 'runExample( "vectoro1")' instead
## But this is useful if you wanna see all the TMBO macros expanded
file.copy( system.file( 'examples/arrayo1.cpp', package='TMBO'),
    './arrayo1.cpp') # avoid messing up package's own folder
compile( './arrayo1.cpp', ppcheck=TRUE)

## End(Not run)
```

MACROS *C-side syntax of TMBO*

Description

TMBO's syntax is so trivially simple that it almost doesn't need this documentation, which looks more scary than it is. But I guess documentation is always good, and if I didn't write my own, I couldn't really complain about other people not doing it properly, which would be frustrating. So here goes.

Basically, to use TMBO features you have to:

1. Slightly modify the start of your TMB code and (some) of its `#include` directives;

2. Slightly change the syntax of (some) variables acquired from R (`DATA_` and `PARAMETER_` statements);

3. Optionally, declare some Ranges acquired from from R, which are a bit like a special type of `DATA`;

3. Declare new TMB variables via eg `MATRIX1( int, z, 3, 3)` instead of `matrix<int> z(3,3)`, or `ARRAYO( Type, nfish, YEARS, SEXES, AGES)` instead of `array<Type> nfish(...)`;

4. Optionally, use the much simpler syntax of `FOR` in place of "simple" for-loops.

5. Optionally, to embed a function within your TMB code, you can use the `FUNCTOID` syntax.

Ranges

If you are only using the 1-based and non-character-indexed aspects of TMBO, just to make your TMB variables start at 1 like their R counterparts, then you don't really need to know this. Just declare variables with eg `MATRIX1(x,Ni,Nj)` and `DATA_IARRAY1(my_nice_R_shaped_array)` etc, and see the section on `FOR`-loops. But it won't overtax most brains to just read this anyway...

A Range in TMBO is really just a pair of integers in parentheses, eg `(7,11)` which "means" all integers from 7 through to 11 inclusive. You can use Ranges only in two places: when declaring new TMB variables via eg `VECTORO` etc; and in simplified for-loops with `FOR`. You can specify a Range "manually", eg `FOR(i,(1,N))` (though `FOR(i,1,N) is simpler and also works), but you can also have named Ranges passed in from R,` eg ICONSEQ(SAMP_YEARS)'. Named ranges are often simpler to mention in loops and further declarations. You can also have "chinds" passed in from R, which are character Ranges like the `names` or `dimnames` of an R variable. Internally in your TMB code, chinds are just Ranges starting from 1, but the information about them is propagated into new variables that use that chind, and then back to R by `REPORT()`. Variables coming in from R via `DATA_MATRIXO` etc may have a chind associated with each of their dimensions— you don't have to do anything explicit to make that happen. The character information associated with a chind— whether it was originally declared explicitly via `CHIND`, or implicitly by association with a `DATA_MATRIX` etc that has chind(s) as its Range(s)— gets used in any variables it's passed onto only when they are `REPORT`ed to R, at which point their names/dimnames are set accordingly.

The Range(s) of an existing variable `x` (one Range per dimension) can be accessed via eg `DIMRANGE(x,3)` for the 3rd dimension, and used in declarations and FOR-loops: eg `VECTORO(y,DIMRANGE(x,3))` and `FOR( j, DIMRANGE(y))`. If `x` is a vector, you can only refer to `DIMRANGE(x)` without the second argument.

Ranges are not genuine variables within TMB, which is why you can only use them within specific TMBO contexts. However, you can access the ends of the range as if they were genuine variables, using eg `FIRSTEL(v)` and `LASTEL(v)` for vector `v`, or `FIRSTEL(ar,3)` for the first index of the 3rd dimension of array `ar`. Thus, `DIMRANGE(ar,3)` is in fact (almost) exactly equivalent to writing `(FIRSTEL(ar,3),LASTEL(ar,3))`.

Functoids

Sometimes you want to embed a function within your TMB objective code, so that it knows about (and can change) your TMB variables directly, without having to pass zillions of them every time you call your embedded function. You can do this by defining a "functoid", as in the "functoid.cpp" example. The raw C++ syntax is not thaaat bad, but it's simpler to just use the word `FUNCTOID`, as per the example.

Because functoids have free & full access to variables that already exist inside your TMB objective (which BTW have to be declared *before* the functoid definition), you often don't need to pass `VMAO`-type arguments explicitly. However, if you do, you need to use the syntax of `USE_VECTORO` (when declaring the functoid) and `PASS_VECTORO` (when calling it), as per the "functoid.cpp" example—analogously for `MATRIXO`, `VECTOR1`, etc.

The only thing you can't do with a functoid, is to *create* a variable inside it that will be visible outside it (i.e., in the main body of your objective, or in other functoids). Just declare anything like that before calling the functoid.

Note that you mustn't try to associate a type with the `FUNCTOID`. It turns out that the actual type of the functoid *always* has to be `auto` "becoz C++", but that's done already for you by using the word `FUNCTOID`.

Pseudo-usage

```
### Declaring variables from R
#
# DATA_VECTORO( x)
# DATA_IVECTORO( x)
# DATA_MATRIXO( x)
# DATA_IMATRIXO( x)
# DATA_ARRAYO( x)
# DATA_IARRAYO( x)
#
# DATA_VECTOR1( x)
# # ... and similarly DATA_IVECTOR1 etc
#
# PARAMETER_VECTORO( x)
# PARAMETER_VECTOR1( x)
# # ... and similarly PARAMETER_MATRIX1 etc
#
# DATA_FACTOR1( x)
#
# ## Declaring new variables
#
# VECTORO( tipe, x, R)
```

```
# MATRIXO( tipe, x, R1, R2)
# ARRAYO( tipe, x, ...)
#
# VECTORO_sameshapeas( tipe, x, template)
# MATRIXO_sameshapeas( tipe, x, template)
# ARRAYO_sameshapeas( tipe, x, template)
#
# VECTOR1( tipe, x, RorN)
# MATRIX1( tipe, x, RorN1, RorN2)
# ARRAY1( tipe, x, ...)
#
# VECTOR1_sameshapeas( tipe, x, template)
# MATRIX1_sameshapeas( tipe, x, template)
# ARRAY1_sameshapeas( tipe, x, template)
#
# # Declaring ranges
#
# ICONSEQ( inds)
# CHIND( ch)
# CHIND_EXPAND( ch, ...)
#
# ## Access to range information
# DIMRANGE( v)
# DIMRANGE( matORar, idim)
# FIRSTEL( vORr)
# LASTEL( vORr)
# FIRSTEL( matORar, idim)
# LASTEL( matORar, idim)
#
# ## FOR-loop simplifiers
#
# FOR( index_name, R)
# FOR( index_name, lower, upper)
# ROF( index_name, upper, lower)
#
# ## Functoids
#
# FUNCTOID fun( ...)
# USE_VECTORO( tipe, x)
# USE_MATRIXO( tipe, x)
# USE_ARRAYO( tipe, x)
# PASS_VECTORO( x)
# PASS_MATRIXO( x)
# PASS_ARRAYO( x)
# USE_VECTOR1( tipe, x)
# USE_MATRIX1(tipe,  x)
# USE_ARRAY1( tipe, x)
# PASS_VECTOR1( x)
```

```
# PASS_MATRIX1( x)
# PASS_ARRAY1( x)
#
# ## Headers and directives
#
# TMB_MAGIC  # instead of objective squiggle blalblah
# TMBO_MAGIC # synonym
#
# HASH_INCLUDE filespec
# HASH_DEFINE ...
# HASH_PRAGMA ...,  HASH_IF ...,  HASH_ENDIF ..., etc
```

Pseudo-arguments

tipe A C++ "type" acceptable to TMB. Usually `Type` or `int`. IDK what else is allowed.

x Name of variable

R, R1, R2 A Range (see below)

RorN, RorN1, RorN2 An integer, chind, or (less likely) a Range that starts at 1.

... in `ARRAYO`, these must be Ranges, like `R` in `VECTORO`. In `ARRAY1`, these must be integers or Ranges, like `RorN` in `VECTOR1`. In `CHIND_EXPAND`, it's all the names of the members of the chind, eg `CHIND_EXPAND( COLOURS, Red, Green, Blue)`, which will lead to integer variables in TMB called `Red_`, `Green_`, and `Blue_` whose values are passed in automatically from R. In `FUNCTOID`, it's the parameter list of your functoid. In the various `HASH_BLAH` macros, it's whatever you would put after the `#blah`, which will happen during the "main" (ie second pass) compilation.

template An existing variable whose dimensions, offsets, and names/dimnames (but not necessarily C++ "type") are to be applied to the new variable `x`. EG suppose you already have a 7-dimensional `DATA_IARRAYO` called `observed_nkinpairs`; you might well want `expected_nkinpairs` of type `Type` with exactly the same dimensions. The nomenclatural minefield around `Type` is not my fault BTW...

v in `DIMRANGE(v)`, a TMBO vector

matORar in `DIMRANGE( matORar, idim)`, a TMBO matrix or array

vORr in `FIRSTEL` and `LASTEL`, either a TMBO vector or a Range

idim in `DIMRANGE`, `FIRSTEL`, and `LASTEL`, which of the dimensions to return

index_name in `FOR`, the name of the integer variable that becomes the loop index.

upper, lower in `ROF`, the loop will step downwards from `upper` to `lower` inclusive. In `FOR` it steps upwards from `lower` to `upper` inclusive.

Technicalities

Directives: `TMBO` uses two-phase compilation, whereby the first phase sets up some variables & macros used in the second phase. Some "standard" C preprocessor directives should only be expanded in the second phase, not the first; examples are including "system" header files (as opposed to bits of code you wrote yourself and are just keeping in a separate file, for which normal `#include` should be used), and perhaps defining some object macros. To make that happen, just write eg `HASH_INCLUDE <systemhead.hpp>` instead of `#include <systemhead.hpp>` etc.

Propagation of chinds: A new variable that is declared using a DIMRANGE from an existing variable, will acquire any chind that was associated with that particular DIMRANGE. However, this requires a bit of trickery, and it's fragile; DIMRANGE(x,2) will work but DIMRANGE(x,1+1) won't propagate the chind (which only matters for post-hoc formatting after REPORT; TMB *calculations* will not be affected). Nor will the otherwise-equivalent (FIRSTEL(x,2),LASTEL(x,2)) give you the (presumably) desired REPORT.

MakeADFun — *Construct objective function and derivatives allowing general index offsets*

Description

TMBO::MakeADFun is just like TMB::MakeADFun, but lets you provide "dimensional" information that can be used (i) by your TMBO code, and (ii) to automatically make nice REPORT() output (ie with index-offsets and dimnames etc). That info mostly comes in the form of the ranges argument— similar to data= and parameters=— which is most easily specified by calling TMBO_ranges, eg:

```
obj <- MakeADFun(
    data= <known things>,
    parameters= <unknown things>,
    ranges=TMBO_ranges( <see below>),
    <more args>
  )
```

TMBO_ranges is "just" a wrapper for list() but is smarter about names and so on, to Save You Effort. TMBO_ranges has deliberately non-standard evaluation, so don't try to be too clever; follow the EXAMPLES.

Ranges in tmbo: Named ranges are useful for code clarity in and between R and TMB(O), though not compulsory (you *can* instead just use DIMRANGE(myar,3) etc in TMBO). Range examples:

```
obj <- MakeADFun( <your stuff>, ranges=TMBO_ranges(
  SAMP_YEARS=1987:2003,          # consecutive integers
  CATCH_YEARS= c( 1980, 2008),   # just a pair (start and end)
  SEXES= c( 'Female', 'Male')    # characters
))
```

Ranges for TMBO on the R side are either integer (numeric is usually fine) or character. Integer ranges can either be consecutive sequences with step +1, eg 1986:1992, or a pair with the first no larger than the second, eg c(1986,1992) or c(1,1) but not c(1,0). On the C side, the corresponding range is declared as eg ICONSEQ(Years);.

Note that most integer variables in R actually end up as numeric, which is normally fine and you don't need to convert explicitly; but be aware that numeric ranges will be coerced to integer with a check that no rounding is taking place. So don't use eg c(5.3, 11.7), is what I'm saying.

Character ranges are declared on the C side as `CHIND( Sexes);` or `CHIND_EXPAND( Sexes, Male, Female);`, although they are "really" integer ranges starting at 1 from TMBO's PoV. They can be used just like integer ranges, in eg `FOR( s, Sexes)` and `VECTORO( int, x, Sexes);` or `VECTOR1( int, x, Sexes)`; see SUBTLE DIFFERENCE below.

The difference between `CHIND` and `CHIND_EXPAND` is that the latter also requires you to name the actual *elements* of the character range, and it turns each into a variable that can be mentioned explicitly in your TMB code. So, if `Sexes=c("Female","Male")` in R, then with `CHIND_EXPAND( Sexes, Male, Female);` in TMB you can write not just `FOR( s, SEXES)` but also eg `myvec( Female_) = 99;` (note the trailing underscore). If you just use `CHIND( Sexes);` then `FOR( s, Sexes)` is fine but you can't refer to `Female_`. Plain `CHIND` is fine if there's no reason to write qualitatively different TMB code for different "levels" of the character range, and is also unavoidable if you don't know in advance what the "levels" will be. `CHIND_EXPAND` makes sure the TMB order matches the R order (ie the order given inside `CHIND_EXPAND` is irrelevant, as long as there's the same names overall), so you don't have to worry about making sure they're the same. If you are using `CHIND_EXPAND` in TMB code, then the R-side information in `TMBO_ranges` should be wrapped in `list`, eg `ranges=TMBO_ranges( list( Sexes), ...)`.

Subtle difference between o and 1: Hopefully I've explained this somewhere else too; it doesn't really belong here, but... Anyway: `VECTOR1`, `MATRIX1`, variables in TMBO are guaranteed to use 1-based indexing. `VECTORO` etc can of course be 1-based too if that's what's specified in their declaration, and there is *no* difference between the behaviour of VMA1 and 1-based VMAO variables within TMBO. The subtle difference *only* applies on R's PoV, and only if/when you `REPORT()` the variable. Then, the 1-based version will return a standard R vector/matrix/array, with the character range turned into `names` or `dimnames`. In contrast, the O-based version will always generate an `offarray`.

Usage

```
MakeADFun( data, parameters, ..., ranges = list())
TMBO_ranges(...)
```

Arguments

`data`, `parameters`	As per `TMB::MakeADFun` (qv).
`ranges`	Optional list of named ranges (for vec/mat/array indices), which can be referenced in TMB code. See **Details**.
`...`	In `MakeADFun`:like `TMB:MakeAdFun` (qv). In `TMBO_ranges`:a set of integer or character ranges, either variables that already exist, or created on-the-fly during this call, and each possibly wrapped in a call to `list()`. See **Details**.

Details

`TMBO_ranges` tries hard to figure out what to call the ranges, to save you the unutterable tedium of eg `Sexes=Sexes`. If you already have a variable called `Sexes`, you can just pass it straight into `TMBO_ranges` without naming it.

For chinds (a character vector used as a range— like a `dimnames` or `names`), you can make its elements (as well as the whole chind) available in TMBO code by wrapping it in `list`, eg `TMBO_ranges(`

..., list(COLOURS)) if COLOURS already exists, or or TMBO_ranges(... list(COLOURS=c("Red", "Green", "Blue")) if it doesn't. The corresponding TMBO declaration needs to be CHIND(COLOURS, Blue, Green, Red), ie naming the (same) elements, but the order can be different— it will Just Work. Elsewhere in your TMBO code, you can then write eg thing(Red_)=99; note that underscore, which shouldn't be in the CHIND statement. Note also that you have to explicitly call list inside the call to TMBO_ranges; you can't predefine the thing as a list and then pass it in, which would count as "trying to be too clever" here. Non-standard evaluation, like I said...

If you don't wrap a chind in list, and/or if in TMBO you just declare it as CHIND(COLOURS)don't then TMBO code can still refer to the entire range COLOURS, when declaring a new variable or in FOR-loops, but cannot mention the individual colours by name.

Value

MakeADFun, like its namesake in package **TMB**, returns a list with components ($fn, $gr, etc) suitable for use with an R optimizer, such as nlminb or optim. You don't need to know what TMBO_ranges returns, because you will only invoke this inside a call to TMBO::MakeADFun. But since you ask: it's a list.

See Also

MakeADFun in package **TMB**.

Examples

```
"Use 'runExample()' to list examples. Then run one of them, and look at its code."
Samp_years <- 2003:2008 # can use either a sequence...
Birth_years <- c( 1960, 1990) # or a start/end pair
Sexes <- c( 'Female', 'Male')
Stages <- c( 'Young', 'Old')
# Next is pretty useless except as 'range' arg in a call to 'MakeADFun'. But...
TMBO_ranges(
  Samp_years,                 # Pre-defined
  Birth_years=c( 1960, 1990), # On-the-fly; also NB start/end pair
  list( Sexes),               # Female_ and Male_ will be variables inside TMBO
  Stages,                     # no Young_ or Old_ available inside TMBO
  Morphs= c( 'Hairy', 'Baldy'),  # On-the-fly
 Lightness= list( Shade= c( 'Dark', 'Light')) # bad, ambiguous; but "Lightness" takes priority
)
```

---

make_all_ELn *Generate array macros*

Description

make_all_ELn creates C header files with repetitive macros needed for TMBO arrays. You probably never need to call it; it's normally run invisibly and once only by the **TMBO** package itself, at installation or first use. If you ever need even-higher-dim arrays than TMBO provides by default

(currently 7), then you can run `make_all_ELn` manually. Now, 7 might sound like a lot, but it is actually not enough for many realistic CKMR models.

Note that TMB itself (as of version 1.9.6) "only" supports arrays of up to 7D. If you want to add more, you can easily do so by modifying `system.file("include/tmbutils/array.hpp",package="TMB")` yourself. Just search for the lines with `int i7` or `int n7`, and add extra functions in the extremely obvious way. (I could write code to do this automatically, but it's not the job of `TMBO` to patch the source of `TMB`!)

Usage

```
make_all_ELn( n, files= TRUE, count_dims= FALSE)
```

Arguments

`n`	Maximum array dimension
`files`	Default `TRUE` should update the header files in package **TMBO** itself. `FALSE` returns the macros themselves, as R character vectors. Otherwise, `files` should be a character vector with the paths to the file(s) to be created. It can be either length-1 or length-2. In either case, two files will be created (because some macros are needed in the first pass, and some in the second). If `length(files)==1`, then the second filename will have be the same as the first with the first occurrence of the digit "1" replaced by "2".
`count_dims`	The newer fancier OOB-checker should also trap attempts to index an array with the wrong number of subscripts (compared to its declaration), which otherwise causes a typical TMB crash. It's not working yet, so default is FALSE! This does require a bit of non-macro actual C code (`_TMBO__dimbo` in "TMBO2.h").

Value

The filenames, or the macros themselves iff `files==FALSE`.

Examples

```
# Boring...
make_all_ELn( 3, FALSE)
```

`runExample` *Run TMBO example*

Description

Compile and run a test example from the **TMBO** package— just like `TMB::runExample` (qv). See the latter for details. To see what examples exist, just do `runExample()`.

If an example doesn't work, try also passing an (unnamed) argument "&> logfile.log" to get the error log. That's on Windows only; Linux syntax is something different (whatever you do to redirect the output of g++.) See also `TMBO::compile` and the `ppdebug` argument, in case of TMBO preprocessing woes.

The result of the example (usually its final line) is returned invisibly (unlike the TMB version), so eg if that's an object from MakeADFun, you can assign it and play around with it further.

Usage

```
runExample(
  name,
  all= FALSE,
  exfolder= NULL,
  subarch= FALSE,
  dontrun= FALSE,
  clean= FALSE,
  ppflags= NULL,
  ...
)
```

Arguments

name	string with *just* the name of the example (no path, no extension). If missing and if all=FALSE, runExample returns the possible values.
all	set TRUE to run 'em all
exfolder	where to look for the example, and where to build it. Defaults to <lib>/TMBO/examples. May result
subarch	obsolete post R4.3; leave it!
dontrun	You might just want to compile it, not run it.
clean	Recompile from scratch?
ppflags	Character vector to be passed to compile, just for the preprocessing step. EG to trap OOBs nicely, "-DTMBOOBOO"; or for example "oob1" where the default is to trap OOBs, "-DFORCE_CRASH" to show what happens if you don't.
...	All args as per TMB::runExample (qv). Any un-named unmatched ones are passed to compile, though it's a bit hard to predict which ones those will be.

Value

The result of the final line of the example (assuming it worked), invisibilized for brevity. If no arguments are given, the names of the available examples are returned.

Examples

```
runExample()
## Not run:
runExample( 'vectoro1')

## End(Not run)
```

Index